COMPUTER SYSTEMS ORGANIZATION

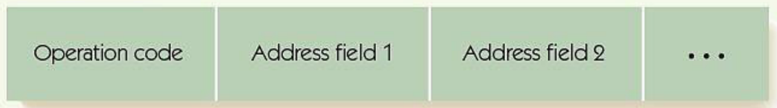
https://xkcd.com/1077/

LEARNING OBJECTIVES

- Control Unit special registers
- List and explain the types of instructions in a typical instruction set, and how instructions are commonly encoded
- Describe the components of a random access memory system, including how fetch and store operations work, and the use of cache memory to speed up access time

AN INSTRUCTION

FIGURE 5.20



Typical machine language instruction format

CONTROL UNIT SPECIAL REGISTERS

Control unit contains:

- **Program counter (PC)** register: holds address of next instruction
- **Instruction register (IR)**: holds encoding of current instruction
- Instruction decoder circuit
  - Decodes op code of instruction and signalshelper circuits.
    - Helpers send addresses to proper circuits
    - Helpers signal ALU, I/O controller, and memory

THE PC

- The Program Counter is part of the control unit and is used to fetch the next instruction
- The bus is the collection of wires which link the components of the architecture together (values: data, instructions and addresses move along the bus)
- The MAR and MDR we have seen, the IR is a register to hold the current instruction while it is being executed

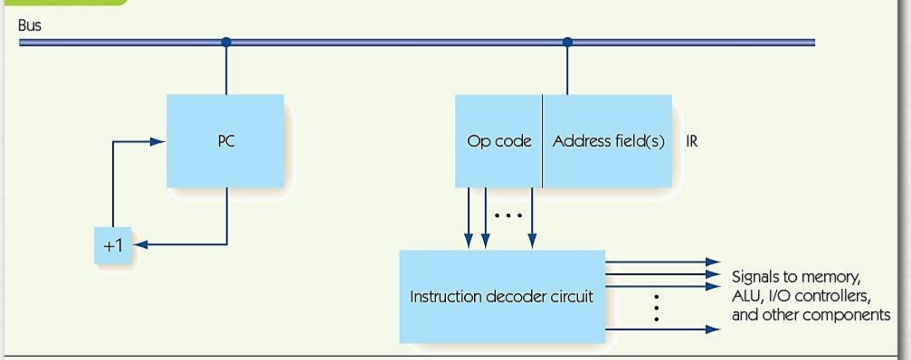
  PC → bus → MAR

  get memory value at MAR → MDR → bus → IR

  PC + 1 → PC   (this happens immediately the value of the PC has been used)
- If we jump to a different location we load the PC with the destination address

PC MEMORY AND BUS

FIGURE 5.22

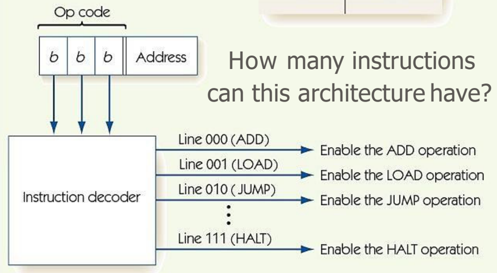

Bus
PC
+1
Op code
Address field(s)
IR
Instruction decoder circuit
Signals to memory, ALU, I/O controllers, and other components

Organization of the control unit registers and circuits

DECODING THE OPCODE

FIGURE 5.23

Op code	Instruction
000	ADD
001	LOAD
010	JUMP
⋮	⋮
111	HALT



How many instructions can this architecture have?

The instruction decoder

MACHINE LANGUAGE AND INSTRUCTION SET

- **Machine language**
  - Binary strings encode instructions
  - Instructions can be carried out by hardware
  - Sequences of instructions encode algorithms
- **Instruction set**
  - Instructions implemented by a particular chip
  - Each kind of processor has a different instruction set (speaks a different language)
  - All do similar things e.g. +, -, ×, ÷, AND, OR, NOT, SHIFT, LOAD, STORE, COMPARE, BRANCH

WE HAVE TO HAVE THESE THINGS

Instruction set examples:

- Data transfer, e.g., move data from memory to register
- Arithmetic, e.g., add, but also AND
- Comparison: compare two values
- Branch: change to a nonsequential instruction
  - Branching allows for conditional and loop forms
  - E.g., JUMPLT *a* = If previous comparison of A and B found A < B, then jump to instruction at address *a*

DATA TRANSFER

- We are going to look at memory shortly
- We don’t calculate "directly" on values in memory
  - Even an instruction which increments a value in memory does
    - load the value into a CPU register
    - increment the value
    - store the value back into its memory location
    - several clock cycles
- We can move (transfer) data
  - from memory to registers (LOAD)
  - from registers to memory (STORE)
  - from registers to registers
  - (and on some architectures from memory to memory)

ARITHMETIC AND LOGIC

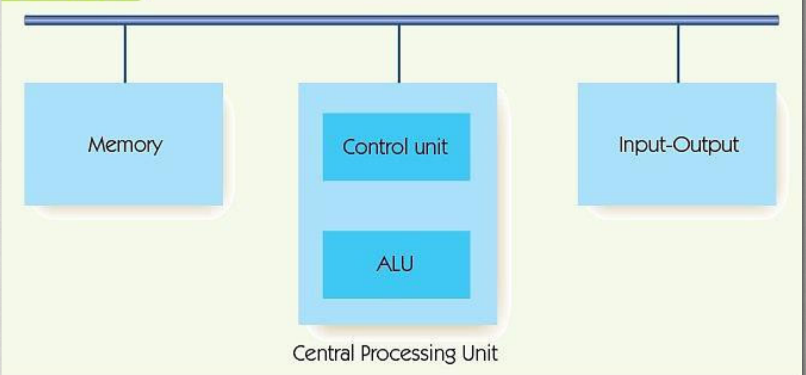
- We at least need an ADD - commonly the 4 arithmetic operations are available (this gets complicated if we include floating point)
  - how can we get by with just an ADD?
- We can do any logical operation using a functionally complete operation such as NAND or NOR
- Also very convenient if we have a RIGHT SHIFT
  - why don’t we really need a LEFT SHIFT if we have ADD?

COMPARISON AND BRANCHING

- In order to make decisions (think "if statements" and "loops")
- We must have a way of comparing values
  - this can be done with special instructions like COMPARE
  - or we can set special bits, like Carry or Overflow as a result of arithmetic instructions e.g. SUBTRACT, and make the decision on the basis of those bits
- We must have a way to go to different *next* instructions depending on the comparison result e.g. JUMPGT
- We will come back to these after looking at memory

VON NEUMANN ARCHITECTURE (MEMORY)

FIGURE 5.3



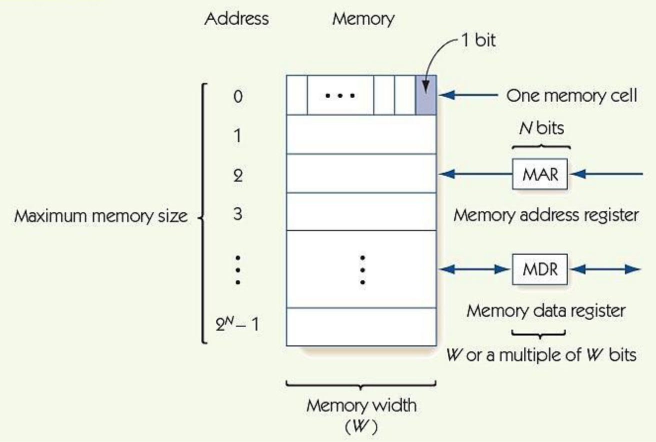
Components of the Von Neumann architecture

MEMORY

- **Memory:** functional unit where data is stored/retrieved
- **Random access memory (RAM)**
  - Organized into **cells**, each given a unique **address**
  - Equal time to access any cell
  - Cell values may be read and changed
- **Read-only memory (ROM):** A type of RAM with prerecorded information that cannot be modified or changed
- **Cell size/memory width** is typically 8 bits (byte addressing)
- **Maximum memory size/address space** is 2^N, where N is length of address

ABSTRACT VIEW OF MEMORY

FIGURE 5.4



Structure of random access memory

MEMORY SIZES

N	Maximum Memory Size (2^N)
16	65,536
20	1,048,576
22	4,194,304
24	16,777,216
32	4,294,967,296
48	281,474,976,710,656
64	18,446,744,073,709,551,616

MEMORY REGISTERS

- **Memory address register (MAR)** holds memory address to access
- **Memory data register (MDR)** receives data from fetch and holds data to be stored
- Fetch (LOAD): retrieve from memory (**non-destructive fetch**)
  1. Load the address into the MAR
  2. Decode the address in the MAR
  3. Copy the contents of that memory location into the MDR
- Store (STORE): write to memory (**destructive store**)
  4. Load the address into the MAR
  5. Load the value into the MDR
  6. Decode the address in the MAR
  7. Copy the contents of the MDR into that memory location
- Is a MOVE from one memory location to another destructive or non-destructive?
- **Memory access time**
  - Time required to fetch/store
  - Modern RAM requires 5-10 **nanoseconds**

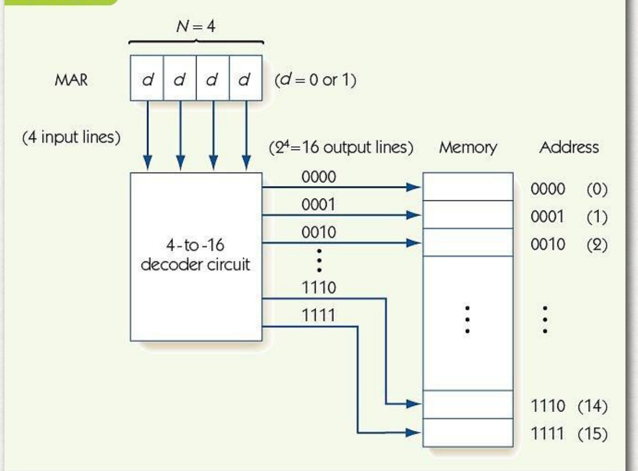
MEMORY CIRCUITS

- Memory system circuits: **decoder** and **fetch/store controller**
- Decoder converts MAR into signal to a specific memory cell
  - One-dimensional versus two-dimensional memory organization
- Fetch/Store controller ▶ traffic cop for MDR
  - Takes in a signal that indicates fetch or store
  - Routes data flow to/from memory cells and MDR

SIMPLE DESIGN (1 DIMENSION)

No good for large memory, too many output lines

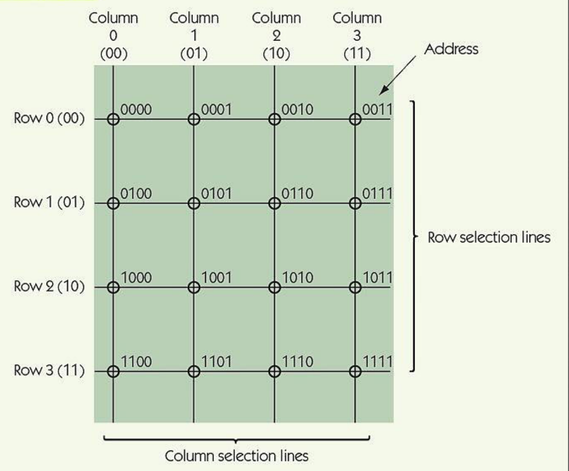
FIGURE 5.6



Organization of memory and the decoding logic

A MORE EFFICIENT DESIGN (ROWS & COLUMNS)

FIGURE 5.7

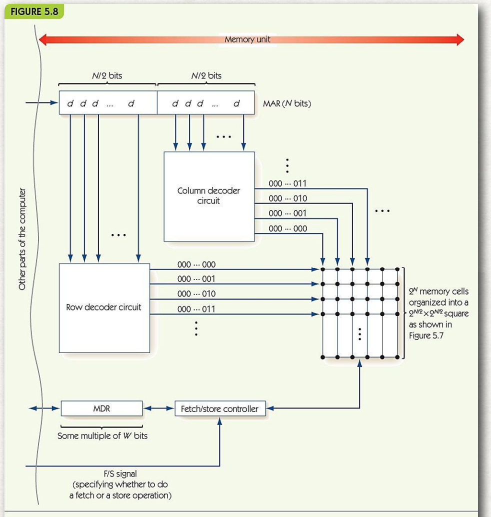


Two-dimensional memory organization

ROW & COLUMN DECODERS

Real RAM controllers are more complex

FIGURE 5.8



Overall RAM organization

If 16 bit address, 256 + 256 lines rather than 65536

CACHE MEMORY

- RAM speeds increased more slowly than CPU speeds (Fig. 5.9)
- **Cache memory** is fast but expensive
  - Built into the CPU for fast access times
- **Principle of locality**
  - Values close to recently accessed memory are more likely to be accessed
  - Load neighbours into cache and keep recent values there
- **Cache hit rate:** percentage of times values are found in cache